Constraint-based and Declarative Analyses

Kristopher Micinski
CIS 700 —Program Analysis: Foundations and Applications
Fall ’19, Syracuse University

$$\{\mathtt{fn\ x\ \texttt{=>}}\ [\mathtt{x}]^1\} \subseteq \widehat{\mathsf{C}}(2)$$
$$\{\mathtt{fn\ y\ \texttt{=>}}\ [\mathtt{y}]^3\} \subseteq \widehat{\mathsf{C}}(4)$$

Today in class, I want to present several more perspectives on 0CFA

These topics can be expanded to account for more precise analyses

Helps to see in simpler context (0CFA) first, as 0CFA often eliminates technical overhead present in higher-precision analyses

Last class we saw 0CFA

Base

$$\big(\lambda(x)\ e\big) \Rightarrow \big(\lambda(x)\ e\big)$$

Calls

$$\frac{\big(\lambda(x)\ e'\big) \Rightarrow e_0 \quad v \Rightarrow e_1}{v \Rightarrow x}\ (e_0\ e_1)$$

Returns

$$\frac{\big(\lambda(x)\ e\big) \Rightarrow e_0 \quad v \Rightarrow e}{v \Rightarrow (e_0\ e_1)}\ (e_0\ e_1)$$

$$\big(\lambda(x)\ e\big) \Rightarrow \big(\lambda(x)\ e\big)$$

$$\frac{\big(\lambda(x)\ e'\big) \Rightarrow e_0 \quad v \Rightarrow e_1}{v \Rightarrow x}$$

$$\frac{\big(\lambda(x)\ e\big) \Rightarrow e_0 \quad v \Rightarrow e}{v \Rightarrow (e_0\ e_1)}$$

We run these “flow implications” until we can’t run them anymore

$$\big((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\big)^7$$

$$\big(\lambda(x)\ e\big) \Rightarrow \big(\lambda(x)\ e\big)$$

$$\frac{\big(\lambda(x)\ e'\big) \Rightarrow e_0 \quad v \Rightarrow e_1}{v \Rightarrow x}$$

$$\frac{\big(\lambda(x)\ e\big) \Rightarrow e_0 \quad v \Rightarrow e}{v \Rightarrow (e_0\ e_1)}$$

We run these "flow implications" until we can't run them anymore

$$\big((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\big)^7$$

$$\left(\lambda(x^1)\ (x^2\ x^3)\right)^4 \Rightarrow \left(\lambda(x^1)\ (x^2\ x^3)\right)^4$$

$$\left(\lambda(y^5)\ (y^6\ y^7)\right)^8 \Rightarrow \left(\lambda(y^5)\ (y^6\ y^7)\right)^8$$

$$\left(\left(\lambda(x^1)\ (x^2\ x^3)\right)^4\ \left(\lambda(y^5)\ (y^6\ y^7)\right)^8\right)^9$$

$$\left(\lambda(x^1)\ (x^2\ x^3)\right)^4 \Rightarrow \left(\lambda(x^1)\ (x^2\ x^3)\right)^4$$
$$\left(\lambda(y^5)\ (y^6\ y^7)\right)^8 \Rightarrow \left(\lambda(y^5)\ (y^6\ y^7)\right)^8$$

$$\left(\lambda(y^5)\ (y^6\ y^7)\right)^8 \Rightarrow x$$

$$\left(\left(\lambda(x^1)\ (x^2\ x^3)\right)^4\ \left(\lambda(y^5)\ (y^6\ y^7)\right)^8\right)^9$$

$$\left(\lambda(x^1)\ (x^2\ x^3)\right)^4 \Rightarrow \left(\lambda(x^1)\ (x^2\ x^3)\right)^4$$

$$\left(\lambda(y^5)\ (y^6\ y^7)\right)^8 \Rightarrow \left(\lambda(y^5)\ (y^6\ y^7)\right)^8$$

$$\left(\lambda(y^5)\ (y^6\ y^7)\right)^8 \Rightarrow x$$

$$\left(\lambda(y^5)\ (y^6\ y^7)\right)^8 \Rightarrow y$$

$$\left(\left(\lambda(x^1)\ (x^2\ x^3)\right)^4\ \left(\lambda(y^5)\ (y^6\ y^7)\right)^8\right)^9$$

How do we actually “solve” these flow equations

Observe: can read them as a collection of constraints over sets

Base

$$\left(\lambda(x)\ e\right) \Rightarrow \left(\lambda(x)\ e\right)$$

Calls

$$\frac{\left(\lambda(x)\ e'\right) \Rightarrow e_0 \quad v \Rightarrow e_1}{v \Rightarrow x}$$

Returns

$$\frac{\left(\lambda(x)\ e\right) \Rightarrow e_0 \quad v \Rightarrow e}{v \Rightarrow (e_0\ e_1)}$$

For each subexpression in the program, we define R(e), called the **flow set** associated w/ that point

$$\big(\lambda(x)\ e\big) \Rightarrow \big(\lambda(x)\ e\big) \qquad\qquad \big(\lambda(x)\ e\big) \in R\big(\lambda(x)\ e\big)$$

$$\frac{\big(\lambda(x)\ e'\big) \Rightarrow e_0 \quad v \Rightarrow e_1}{v \Rightarrow x} \qquad \frac{\big(\lambda(x)\ e'\big) \in R(e_0) \quad v \in R(e_1)}{v \in R(x)}$$

$$\frac{\big(\lambda(x)\ e\big) \Rightarrow e_0 \quad v \Rightarrow e}{v \Rightarrow (e_0\ e_1)} \qquad \frac{\big(\lambda(x)\ e\big) \in R(e_0) \quad v \in R(e)}{v \in R\big((e_0\ e_1)\big)}$$

So we have the following schema for constraints for 0CFA

$$\big(\lambda(x)\ e\big) \in R\big(\lambda(x)\ e\big)$$

$$\big(\lambda(x)\ e'\big) \in R(e_0) \quad v \in R(e_1) \longrightarrow v \in R(x)$$

$$\big(\lambda(x)\ e\big) \in R(e_0) \quad v \in R(e) \longrightarrow v \in R\big((e_0\ e_1)\big)$$

```
(define (constraint? constraint)
  (match constraint
    [`(⊆ ,s0 ,s1) #t]
    [`(∈ ,element ,st) #t]
    [`(← (⊆ ,s0 ,s1) ,(? constraint? bodies) ...) #t]
    [`(← (∈ ,e ,s) ,(? constraint? bodies) ...) #t]
    [else #f]))

(define (expr? e)
  (match e
    [(? symbol? x) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [`(lambda (,(? symbol? x)) ,(? expr? e-body)) #t]
    [else #f]))
```

A tiny constraint solver …

```
(define (solve-constraints constraints)
  (define (body-true body h)
    (match body
      [`(⊆ ,s0 ,s1) (subset? (hash-ref h s0 (set)) (hash-ref h s1 (set)))]
      [`(∊ ,e ,st) (set-member? (hash-ref h st (set)) e)]))
  (define (iter h)
    (let ([next
           (foldl
            (lambda (constraint h)
              (match constraint
                [`(∊ ,e ,s) (hash-set h s (set-add (hash-ref h s (set)) e))]
                [`(⊆ ,s0 ,s1) (hash-set h s1 (set-union (hash-ref h s0 (set)) (hash-ref h s1 (set))))]
                [`(← (⊆ ,s0 ,s1) ,(? constraint? bodies) ...)
                 (if (andmap (lambda (body) (body-true body h)) bodies)
                     (hash-set h s1 (set-union (hash-ref h s0 (set)) (hash-ref h s1 (set))))
                     h)]
                [`(← (∊ ,e ,s) ,(? constraint? bodies) ...)
                 (if (andmap (lambda (body) (body-true body h)) bodies)
                     (hash-set h s (set-add (hash-ref h s (set)) e))
                     h)]))
            h
            (set->list constraints))])
      (if (equal? next h) h (iter next))))
  (iter (hash)))
```

((lambda (x) x) (lambda (y) y))

```
(pretty-print
 (solve-constraints
  (set `(∈ (lambda (x) x) (flow-set (lambda (x) x)))
       `(∈ (lambda (y) y) (flow-set (lambda (y) y)))
       `(← (∈ (lambda (x) x) (flow-set x))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (y) y) (flow-set x))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (x) x) (flow-set y))
           (∈ (lambda (y) y) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (y) y) (flow-set y))
           (∈ (lambda (y) y) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (x) x) (flow-set ((lambda (x) x) (lambda (y) y))))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set x)))
       `(← (∈ (lambda (y) y) (flow-set ((lambda (x) x) (lambda (y) y))))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set x))))))
```

```
((lambda (x) x) (lambda (y) y))
```

Treat rule as schema…

$$\big(\lambda(x)\ e\big) \in R(e_0)\ \ v \in R(e) \ \longrightarrow\ v \in R\big((e_0\ e_1)\big)$$

Generate a separate constraint for each syntactic lambda and application in the program.

```
(pretty-print
 (solve-constraints
  (set `(∈ (lambda (x) x) (flow-set (lambda (x) x)))
       `(∈ (lambda (y) y) (flow-set (lambda (y) y)))
       `(← (∈ (lambda (x) x) (flow-set x))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (y) y) (flow-set x))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (x) x) (flow-set y))
           (∈ (lambda (y) y) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (y) y) (flow-set y))
           (∈ (lambda (y) y) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (x) x) (flow-set ((lambda (x) x) (lambda (y) y))))
```

```
((lambda (x) x) (lambda (y) y))
```

Treat rule as schema…

$$\big(\lambda(x)\ e\big) \in R(e_0) \quad v \in R(e) \quad \longrightarrow \quad v \in R\big((e_0\ e_1)\big)$$

Generate a separate constraint for each syntactic lambda and application in the program.

Can do this better if we use a different language!

```
(pretty-print
 (solve-constraints
  (set `(∈ (lambda (x) x) (flow-set (lambda (x) x)))
       `(∈ (lambda (y) y) (flow-set (lambda (y) y)))
       `(← (∈ (lambda (x) x) (flow-set x))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (y) y) (flow-set x))
           (∈ (lambda (x) x) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (x) x) (flow-set y))
           (∈ (lambda (y) y) (flow-set (lambda (x) x)))
           (∈ (lambda (x) x) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (y) y) (flow-set y))
           (∈ (lambda (y) y) (flow-set (lambda (x) x)))
           (∈ (lambda (y) y) (flow-set (lambda (y) y))))
       `(← (∈ (lambda (x) x) (flow-set ((lambda (x) x) (lambda (y) y))))
```

Datalog

Datalog is a logic programming language that enables chain-forward logical inference via rules

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

Datalog

Equally correct is…

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- edge(x,y),path(y,z).
```

Transitive closure implemented via Datalog

Facts are atomically “true" statements

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

Transitive closure implemented via Datalog

Rules are inductively defined relations

Facts are atomically “true" statements

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

Datalog works by filling up “tables” of facts

Edge	
Column 0	Column 1
0	1
1	2

Path	
Column 0	Column 1

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

To start the process, facts are loaded into tables

Datalog works by filling up “tables” of facts

Edge

Column 0	Column 1
0	1
1	2

Path

Column 0	Column 1

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

Computation evolves according to rules

Datalog works by filling up "tables" of facts

Edge	
Column 0	Column 1
0	1
1	2

Path	
Column 0	Column 1
0	1
1	2

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

Computation evolves according to rules

Datalog works by filling up "tables" of facts

Edge	
Column 0	Column 1
0	1
1	2

Path	
Column 0	Column 1
0	1
1	2
0	2

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

```
edge(0,1).
edge(1,2).
path(x,y) :- edge(x,y).
path(x,z) :- path(x,y),path(y,z).
```

These rules can be compiled to **relational algebra**

Relational algebra:

- Projections (take columns x,y, and z from relation R)
- Renamings (rename column x to column y)
- Joins (matches tuples)

Let's take this rule…

```
path(x,z) :- path(x,y),path(y,z).
```

Compiling to relational algebra:

- Need to join "path" on itself
  - Want pairs of tuples (x,y), (y,z) such that the y matches
  - This can be implemented as a combo rename+join
  - Then putting the result into the path relation

In general, Datalog engines compile to relational algebra, and use optimizations to cull redundant rules / inline appropriately

Crucially, Datalog is **extremely fast**

Fastest program analyses in world currently implemented using Datalog (Smaragdakis et al.)

Relational algebra can be implemented efficiently on GPUs, MPI, and (**recently!**) supercomputers!

Soufflé

Souffle is an:
- Implementation of Datalog
- Super-fast single-node implementation of Datalog

```
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path

edge(0,1).
edge(1,2).

path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

The “.output” command tells Soufflé to write the output to disk

Can also use “.input” to read input tables from disk

Let's think back to our rules…

$$\big(\lambda(x)\ e\big) \in R\big(\lambda(x)\ e\big)$$

$$\big(\lambda(x)\ e'\big) \in R(e_0) \quad v \in R(e_1) \longrightarrow v \in R(x)$$

$$\big(\lambda(x)\ e\big) \in R(e_0) \quad v \in R(e) \longrightarrow v \in R\big((e_0\ e_1)\big)$$

How can we represent this in Datalog?

First attempt…

$$\textbf{Reaches}\big((\lambda(x)\ e), (\lambda(x)\ e)\big)$$

First attempt…

$$\textbf{Reaches}\big((\lambda(x)\ e), (\lambda(x)\ e)\big)$$

Unfortunately, this doesn't work: plain Datalog requires tuples contain atoms (e.g., numbers)

First attempt…

$$\textbf{Reaches}\Big((\lambda(x)\ e), (\lambda(x)\ e)\Big)$$

Unfortunately, this doesn't work: plain Datalog requires tuples contain atoms (e.g., numbers)

Idea: number each subexpression

$$\Big((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\Big)^7$$

Decompose AST into facts

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// ((lambda (x^10) x^2)^3 (lambda (y^11) y^5)^6)^7
// Assume that variable x is named 10 and y is named 11
sourceLambda(3,10,2).
sourceLambda(6,11,5).
sourceVarRef(2,10).
sourceVarRef(5,11).
sourceApplication(7,3,6).
```

$$\left((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\right)^7$$

Decompose AST into facts

Assign each subexpression an ID, linearize into facts that act as constructors. Linked via IDs

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// ((lambda (x^10) x^2)^3 (lambda (y^11) y^5)^6)^7
// Assume that variable x is named 10 and y is named 11
sourceLambda(3,10,2).
sourceLambda(6,11,5).
sourceVarRef(2,10).
sourceVarRef(5,11).
sourceApplication(7,3,6).
```

$$\left((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\right)^7$$

Decompose AST into facts

Assign each subexpression an ID, linearize into facts that act as constructors. Linked via IDs

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// ((lambda (x^10) x^2)^3 (lambda (y^11) y^5)^6)^7
// Assume that variable x is named 10 and y is named 11
sourceLambda(3,10,2).
sourceLambda(6,11,5).
sourceVarRef(2,10).
sourceVarRef(5,11).
sourceApplication(7,3,6).
```

`sourceVarRef(id,varName)` says "the subexpression numbered `id` is a reference to variable `varName`"

$$\left((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ \boxed{y^5})^6\right)^7$$

Decompose AST into facts

Assign each subexpression an ID, linearize into facts that act as constructors. Linked via IDs

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// ((lambda (x^10) x^2)^3 (lambda (y^11) y^5)^6)^7
// Assume that variable x is named 10 and y is named 11
sourceLambda(3,10,2).
sourceLambda(6,11,5).
sourceVarRef(2,10).
sourceVarRef(5,11).
sourceApplication(7,3,6).
```

`sourceLambda(id,varName,bodyId)` says "the subexpression numbered `id` is a lambda w/variable `varName` and whose body is identified by bodyId"

$$\left((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\right)^7$$

Decompose AST into facts

Assign each subexpression an ID, linearize into facts that act as constructors. Linked via IDs

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// ((lambda (x^10) x^2)^3 (lambda (y^11) y^5)^6)^7
// Assume that variable x is named 10 and y is named 11
sourceLambda(3,10,2).
sourceLambda(6,11,5).
sourceVarRef(2,10).
sourceVarRef(5,11).
sourceApplication(7,3,6).
```

`sourceApplication(id,fn,ar)` says "the subexpression numbered `id` is an application whose function is identified by `fn` and whose arg is identified by `ar`."

$$\left((\lambda(x^1)\ x^2)^3\ (\lambda(y^4)\ y^5)^6\right)^7$$

Decompose AST into facts

Assign each subexpression an ID, linearize into facts that act as constructors. Linked via IDs

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// ((lambda (x^10) x^2)^3 (lambda (y^11) y^5)^6)^7
// Assume that variable x is named 10 and y is named 11
sourceLambda(3,10,2).
sourceLambda(6,11,5).
sourceVarRef(2,10).
sourceVarRef(5,11).
sourceApplication(7,3,6).
```

In general, we will assume that the fact tables have been created for us.

As a preprocessing pass, we can generate these rules.

```
;; Populate tables for a term
(define (make-tables expr)
  (match expr
    [(? symbol? x)
     (let ([id (newid)])
       (hash-set! source-var-ref id `(,id ,(lookup-or-new x)))
       id)]
    [`(,e0 ,e1)
     (let* ([id (newid)]
            [fnid (make-tables e0)]
            [argid (make-tables e1)])
       (hash-set! source-application id `(,id ,fnid ,argid))
       id)]
    [`(lambda (,x) ,e-body)
     (let* ([id (newid)]
            [varid (lookup-or-new x)]
            [body-id (make-tables e-body)])
       (hash-set! source-lambda id `(,id ,varid ,body-id))
       id)]))
```

```
;; Tables
(define source-lambda (make-hash))
(define source-var-ref (make-hash))
(define source-application (make-hash))

;; Ids
(define var->id (make-hash))
(define/contract (lookup-or-new var)
  (any/c . -> . number?)
  (if (hash-has-key? var->id var)
      (car (hash-ref var->id var))
      (let ([id (newid)])
        (hash-set! var->id var `(,id ,var))
        id)))

(define x 0)
(define (newid) (let ([id x]) (set! x (add1 x)) id))

(define (expr? e)
  (match e
    [(? symbol? x) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [`(lambda (,x) ,(? expr? e-body)) #t]))
```

```
(define (dump-table table name)
  (for ([tuple (hash-values table)])
    (displayln
      (format "~a(~a)."
              name
              (string-join (map number->string tuple) ",")))))

;(make-tables `((lambda (x) x) (lambda (y) y)))
(make-tables `((lambda (x) (x x)) (lambda (y) (y y))))

(dump-table source-lambda "sourceLambda")
(dump-table source-var-ref "sourceVarRef")
(dump-table source-application "sourceApplication")
```

```
// The flowsTo relation
.decl valueFlowsTo(value:number, expr:number)
.output valueFlowsTo
```

We can now build `valueFlowsTo(v,e)`

In the abstract world, valueFlowsTo relates an abstract value (a lambda, for 0CFA) and an expression.

Here, values will be represented by numbers that identify lambdas in the source program.

Expressions will be represented by numbers that identify subexpressions in the source program.

```
// The flowsTo relation
.decl valueFlowsTo(value:number, expr:number)
.output valueFlowsTo
```

Upshot: valueFlowsTo is a relation between numbers

We can now build `valueFlowsTo(v,e)`

In the abstract world, valueFlowsTo relates an abstract value (a lambda, for 0CFA) and an expression.

Here, values will be represented by numbers that identify lambdas in the source program.

Expressions will be represented by numbers that identify subexpressions in the source program.

Now we must translate each of our rules…

$$\big(\lambda(x)\ e\big) \in R\big(\lambda(x)\ e\big)$$

$$\big(\lambda(x)\ e'\big) \in R(e_0) \quad v \in R(e_1) \longrightarrow v \in R(x)$$

$$\big(\lambda(x)\ e\big) \in R(e_0) \quad v \in R(e) \longrightarrow v \in R\big((e_0\ e_1)\big)$$

$$\big(\lambda(x)\ e\big) \in R\big(\lambda(x)\ e\big)$$

```
valueFlowsTo(lambdaId,lambdaId) :- sourceLambda(lambdaId,_,_).
```

$$\left(\lambda(x)\ e'\right) \in R(e_0) \quad v \in R(e_1) \longrightarrow v \in R(x)$$

```
valueFlowsTo(argVal,x) :- sourceApplication(_,fnId,argId),
                          valueFlowsTo(_,argId),
                          sourceLambda(fnId,x,_),
                          valueFlowsTo(argVal,argId).
```

$$\big(\lambda(x)\ e\big) \in R(e_0) \quad v \in R(e) \quad \longrightarrow \quad v \in R\big((e_0\ e_1)\big)$$

```
valueFlowsTo(v,exprId) :- sourceVarRef(exprId,var),
                          valueFlowsTo(v,var).

valueFlowsTo(v,exprId) :- sourceApplication(exprId,fnId,_),
                          valueFlowsTo(lambdaId,fnId),
                          sourceLambda(lambdaId,_,bodyId),
                          valueFlowsTo(v,bodyId).
```

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

 // The flowsTo relation
 .decl valueFlowsTo(value:number, expr:number)
 .output valueFlowsTo

 valueFlowsTo(lambdaId,lambdaId) :- sourceLambda(lambdaId,_,_).

 valueFlowsTo(argVal,x) :- sourceApplication(_,fnId,argId),
                           valueFlowsTo(_,argId),
                           sourceLambda(fnId,x,_),
                           valueFlowsTo(argVal,argId).

 valueFlowsTo(v,exprId) :- sourceVarRef(exprId,var),
                           valueFlowsTo(v,var).

 valueFlowsTo(v,exprId) :- sourceApplication(exprId,fnId,_),
                           valueFlowsTo(lambdaId,fnId),
                           sourceLambda(lambdaId,_,bodyId),
                           valueFlowsTo(v,bodyId).
```

```
.decl sourceLambda(exprId:number, varName:number, bodyId:number)
.decl sourceVarRef(exprId:number, varName:number)
.decl sourceApplication(exprId:number, fnId:number, argId:number)

// Here's the omega combinator
sourceLambda(6,7,8).
sourceLambda(1,2,3).
sourceVarRef(5,2).
sourceVarRef(4,2).
sourceVarRef(10,7).
sourceVarRef(9,7).
sourceApplication(8,9,10).
sourceApplication(0,1,6).
sourceApplication(3,4,5).
```

```
  // The flowsTo relation
  .decl valueFlowsTo(value:number, expr:number)
  .output valueFlowsTo

  valueFlowsTo(lambdaId,lambdaId) :- sourceLambda(lambdaId,_,_).

  valueFlowsTo(argVal,x) :- sourceApplication(_,fnId,argId),
                            valueFlowsTo(_,argId),
                            sourceLambda(fnId,x,_),
```

```
souffle 0cfa-flow.dl
```

Which generates valueFlowsTo.csv!

valueFlowsTo

1	**1**
6	2
6	4
6	5
6	6